Environment Reuse in the WAM

Bart Demoen[1] and Phuong-Lan Nguyen[2]

[1] Department of Computer Science, K.U.Leuven, Belgium
[2] Institut de Mathématiques Appliquées, UCO, Angers, France
bmd@cs.kuleuven.be, nguyen@ima.uco.fr

Abstract. The TOAM reuses eagerly allocated stack frames, while the WAM avoids to allocate environments. This is investigated by using the tak/4 benchmark as an inital case study for better understanding what one can expect from environment reuse for deterministic predicates in the WAM. Additionally, artificial programs are used to amplify the findings. The experiment compares the impact of reusing an environment versus avoiding to create it: the latter seems a superior technique.

1 Introduction

We assume familiarity with Prolog [1], the WAM [2,3] and the TOAM [4]. Acquaintance with the B-Prolog implementation of the TOAM and with hProlog [5] can also help.

The TOAM and the WAM treat stack frames/environments differently: the TOAM allocates eagerly a stack frame for a predicate, and reuses it when possible. The WAM avoids to allocate an environment for a predicate call, e.g., in case the selected clause is a fact, but possible needs to allocate many environments for the same predicate. This difference stems from different design choices in the two abstract machines and their compilers: the TOAM compiles a predicate at a time, and passes arguments through the stack. The WAM compiles clauses at a time[1], and passes the arguments through a fixed set of argument registers. One can argue endlessly about which is better. We take here a different approach: we investigate to what extent the WAM can benefit from environment reuse, and how effective the environment avoidance optimization, which is in fact known in classical compiler literature as a leaf procedure optimization, compared to environment reuse. We start by using tak/4 as a case study in Section 2. The experiment indicates that the WAM approach can be improved in principle by adopting an environment reuse schema as in the TOAM, at least for tak/4. Section 3 discusses the dynamics of tak/4 and provides a more general insight in the experimental data. Section 4 uses artificial benchmarks for showing the relative merit of environment reuse versus environment avoidance. Section 5 concludes.

The experiments were done on a 1.8 GHz Pentium 4 with Linux (hProlog 2.7 and B-Prolog 7.1b3.2) and on an Intel Mac (hProlog 2.7 and B-Prolog 7.0). The versions of B-Prolog use the TOAM Jr. [6]. Timings are always in milliseconds.

[1] Except for the glue code for indexing.

M. Garcia de la Banda and E. Pontelli (Eds.): ICLP 2008, LNCS 5366, pp. 698–702, 2008.

2 Tak/4 and Its Abstract Machine Code

Below is the source code for tak/4.

```
tak(X,Y,Z,A):-
        (X =< Y ->
            Z = A
        ;
            X1 is X - 1, tak(X1,Y,Z,A1),
            Y1 is Y - 1, tak(Y1,Z,X,A2),
            Z1 is Z - 1, tak(Z1,X,Y,A3),
            tak(A1,A2,A3,A)
        ).
```

The original tak/4 code uses two clauses, but the hProlog compiler and the B-Prolog compiler effectively transform them to the version above with if-then-else. The code generated by B-Prolog and hProlog can be obtained by using *'bpc'/1* and *print_code/1* respectively. A more high level description of the code follows:

```
B-Prolog                                hProlog
========                                =======
@tak:                                   @tak:
      allocate_det
@afteralloc:
      if (! X =< Y) goto @else                if (! X =< Y) goto @else
      unify(Z,A)                              unify(Z,A)
      return_det                              proceed
@else:                                  @else:
                                              allocate
                                              move (X,Y,Z,A) to environment
      X1 is X - 1, tak(X1,Y,Z,A1),            X1 is X - 1, tak(X1,Y,Z,A1),
      Y1 is Y - 1, tak(Y1,Z,X,A2),            Y1 is Y - 1, tak(Y1,Z,X,A2),
      Z1 is Z - 1, tak(Z1,X,Y,A3),            Z1 is Z - 1, tak(Z1,X,Y,A3),
      move (A1,A2,A3) to (X,Y,Z)              load (A1,A2,A3,A) from env
      goto @afteralloc                        deallocate
                                              goto @tak
```

B-Prolog has clearly taken the *allocate* out of the tak loop.

hProlog performs slightly less instruction compression than B-Prolog, it executes some extra instructions for dealing with the argument registers, and it never reuses an environment. Still, hProlog is faster by 33% on the Linux machine, and about 9% on the Mac. This was measured by repeating the goal *tak(18,12,6,_)* 100 times: the first two columns of the table in Section 3 show the figures. The next section explains why the WAM approach works so well.

3 The Dynamics of Tak/4

During one run of the query, the then-branch is taken 47.707 times, while the else-branch is taken 15.902 times: that is (close to) 3 times less. So, in total,

B-Prolog allocates 47.707 times an enviroment, while hProlog does the same 3 times less. The factor 3 results from the fact that 3 out of 4 calls in the body are non-tail calls. One can also see this by considering the execution tree for tak/4: each call-node has outgoing degree equal to 4. Its leaves correspond to calls of the form *tak(X,Y,Z,A)* in which $X \leq Y$, for which the WAM does not allocate an environment. Since the number N of nodes relates to the number I of internal nodes by the simple formula $N - 1 = 4 * I$, the conclusion follows. It is easy to generalize these findings, at least for deterministic programs.

Seemingly, the eager allocation of a stack frame (for deterministic programs) is counterproductive, and it would be a nice experiment to modify B-Prolog to do lazy stack frame allocation, as the WAM does. Since the source code of B-Prolog is not available to us, we have taken the other path: we have modified hProlog to *reuse* its environments, first in the tak/4 benchmark, and later in some artificial benchmarks.

hProlog had already enough instructions to generate code that performs both lazy allocation and environment reuse. The resulting code for tak/4 is:

```
hProlog+reuse
=============
@tak:
      if (! X =< Y) { allocate;
                      move (X,Y,Z,A) to environment;
                      goto @else }
      unify(Z,A)
      proceed
@allocated:
      if (! X =< Y) goto @else
      unify(Z,A)
      deallocate
      proceed
@else:
      X1 is X - 1, tak(X1,Y,Z,A1),
      Y1 is Y - 1, tak(Y1,Z,X,A2),
      Z1 is Z - 1, tak(Z1,X,Y,A3),
      move (A1,A2,A3) to environment slots (X,Y,Z)
      goto @allocated
```

	hProlog	B-Prolog	hProlog +reuse
tak on Linux	315	473	278
tak on Mac	375	412	367

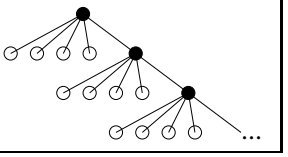
The table above shows the timings for B-Prolog, hProlog and the hProlog version with environment reuse. There is a clear gain in re-using the environment for hProlog, although it depends on the combination of the platform and the gcc version. The above code avoids the allocation of environments and at the same time reuses environments when possible, albeit at the cost of some seemingly duplicate code. Note however that the first $X =< Y$ takes its arguments from the argument registers, while the second takes them from the environment.

It seems clear that the hybrid compilation schema that combines environment avoidance with environment reuse is worth investigating further.

4 Artificial Benchmarks

In order to amplify the potential advantage of environment reuse and environment avoidance, we have constructed a set of benchmarks with a characteristic similar to tak/4, but from which the fluff was removed. We defined predicates $taklike_n$ for $n = 1..10$. As an example, $taklike_5$ is defined as:

```
taklike_5(X) :-
        (X =< 1 -> true                    s.
        ;
           s, s, s, s, s,  % 5 calls to s
           X1 is X - 1, taklike_5(X1)
        ).
```



The execution tree is also shown: the black nodes correspond to calls that can reuse the current environment. The other nodes cannot. For $taklike_n$, the ratio between the calls that can avoid an environment and the calls that can reuse the environment is $n : 1$.

The goal is always of the form $? - taklike_n(5000000)$. The table below shows the timings for hProlog and B-Prolog on two platforms: the upper half on the Linux machine, the seond one on the Mac.

The first (and fifth) row shows the results of running hProlog unaltered on the benchmarks. The second (and sixth) row shows the effect of making hProlog allocate an environment for the *s.* fact: normally the WAM (and hProlog alike) generates just a *proceed* instruction; in this case, hProlog was made to generate an *allocate, deallocate_proceed* sequence, mimicking an eager allocation. The third (and seventh) row shows the result for hProlog with reuse of the environment for the tail call to *tak_like*.

	1	2	3	4	5	6	7	8	9	10	avi
hProlog	150	250	418	450	478	514	542	586	590	630	53
hProlog+extra env	208	458	542	628	714	798	888	972	1056	1148	104
hProlog+reuse	110	208	374	404	432	462	500	524	558	589	53
B-Prolog	226	492	596	692	792	894	994	1100	1204	1304	119
hProlog	183	199	229	252	369	398	422	454	484	492	34
hProlog+extra env	249	325	496	572	670	739	817	899	980	1058	89
hProlog+reuse	123	147	170	196	317	340	364	397	417	444	35
B-Prolog	254	466	590	704	826	959	1154	1191	1367	1456	133

The table also indicates the average increment between successive values of n. It is interesting to see that the B-Prolog figures are the closest to hProlog+extra env. The hProlog columns further show that

- the relative gain of environment reuse depends on n: the gain is larger with smaller n; this gain goes from 26% to 6.5% (on Linux) and 27% to 9.7% (on Mac); note that those are overestimates of what can be achieved in practical programs, because the artificial benchmarks contain hardly any fluff.

– the relative loss of creating the extra environment is about 27% to 45% (on Linux) and 26% to 53% (on Mac); again, those figures overestimate the relative effect.

When one considers the absolute figures (for Linux), one sees that environment avoidance reduces the runtime by 58 msecs up to 518 msecs. Environment reuse gives an almost constant gain between 30 and 40 msecs.

One can conclude that the WAM optimization of not allocating an environment for a fact, is more effective that the TOAM optimization of reusing the stack frame. Both the absolute and relative figures suggest that. Moreover, as expected, environment avoidance becomes better when there are more goals in the body.

5 Conclusion

Tak/4 lends itself easily to environment reuse in the WAM: such reuse is more difficult if a predicate has more than one clause with an allocate, and if one still wants to compile clauses in isolation, as the WAM does. So we cannot claim that we have ultimate answers and solutions. We observed that environment reuse for tak/4 was quite effective, but depending on the platform-gcc combination. The analysis of the experimental results shows that environment avoidance is a better optimization than environment reuse. Of course, for performance reasons, one would like to have them both. The next step should be an adaptation of the hProlog compiler to exploit the reuse of environments.

Acknowledgements

Bart Demoen thanks Research Foundation-Flanders (FWO-Vlaanderen) for support. Part of this work was performed during a visit to IMA, UCO, Angers.

References

1. Clocksin, W., Mellish, C.: Programming in Prolog. Springer, Heidelberg (1984)
2. Aït-Kaci, H.: The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report (1990)
3. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Report 309, SRI (1983)
4. Zhou, N.F.: Global optimizations in a Prolog compiler for the TOAM. Journal of Logic Programming 15(4), 275–294 (1993)
5. Demoen, B., Nguyen, P.L.: So many WAM variations, so little time. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 1240–1254. Springer, Heidelberg (2000)
6. Zhou, N.F.: A Register-Free Abstract Prolog Machine with Jumbo Instructions. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 455–457. Springer, Heidelberg (2007)